CRYPTBARA:
Dependency-Guided Detection of Python Cryptographic API Misuses

Seogyeong Cho, Seungeun Yu, Seunghoon Woo.
Korea University

ASE 2025

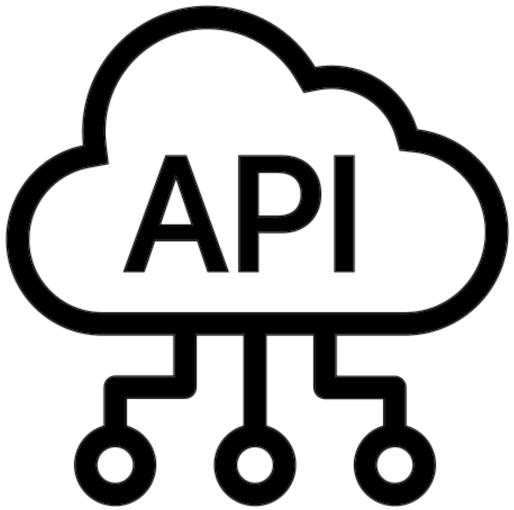
KOREA UNIVERSITY

SSP LAB
Software Security and Privacy Laboratory

Background

Why correct cryptography use matters

- Modern software relies on cryptography to protect confidentiality, integrity, and authenticity
- Cryptography API misuse is a major source of security vulnerabilities



Background

Real-World Example: Weak PBKDF2 Iterations

Listing 4: Real-world misuses patched by our report.

```
def _generate_key_from_password(
  password: bytes, salt: Optional[Union[tr, bytes]] = None
):
  if salt is None:
    salt = os.urandom(16)
  elif isinstance(salt, str):
    salt = salt.encode()
  kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(), length=32, salt=salt,
-   iterations=100000,
+   iterations=800000,
  )
  key = base64.urlsafe_b64encode(kdf.derive(password))
  return key, salt
```

Even if the API is used correctly, weak settings can still create vulnerabilities.

Background

Why Python amplifies the risk

Dynamic features

- Object-dependent meaning
- Indirect construction
- Aliases/imports

In Python, dynamic features make risky patterns more common and more subtle

Challenge

Syntactic Ambiguity **Semantic Ambiguity**

Challenge

Syntactic Ambiguity

What you see at the call site ≠ the actual value or object

A. Context fragmentation

Params (key/IV/iterations) are built across helpers/returns

```
def get_iterations():
  return 10000
def derive_key():
  iters = get_iterations()
  return pbkdf2_hmac(‘sha256’, b‘pass’, b‘salt’, iters)
```

B. Context-dependent resolution

Same method name, different meaning by receiver type

```
def make_crypto():
  return AES.new(b"key1234567", AES.MODE_CBC)
cipher = make_crypto()
cipher.update(b"secret")
```

Challenge

Semantic Ambiguity

Syntactically fine, but safety depends on policy and intent

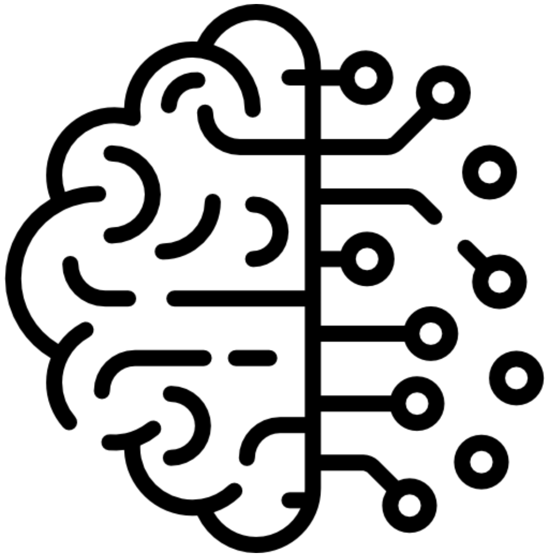
```python
def derive_key(input):
  return ... # generate key from input
def get_key(user_input):
  if user_input:
    return derive_key(user_input)
  return "default_key"
def encrypt(msg):
  user_input = input("Enter (leave blank to use default): ")
  key = get_key(user_input)
  cipher = AES.new(key.encode(), AES.MODE_ECB)
  return cipher.encrypt(msg)
```

Unsafe if a default key is chosen at runtime

Motivation

Make LLMs see the context, not just the line

Structured Dependency Information



Context-aware Judgment

✓ **Solves Syntactic Ambiguity**

✓ **Enables Semantic Judgment**

LLMs alone can’t handle Python’s ambiguity → accuracy drops

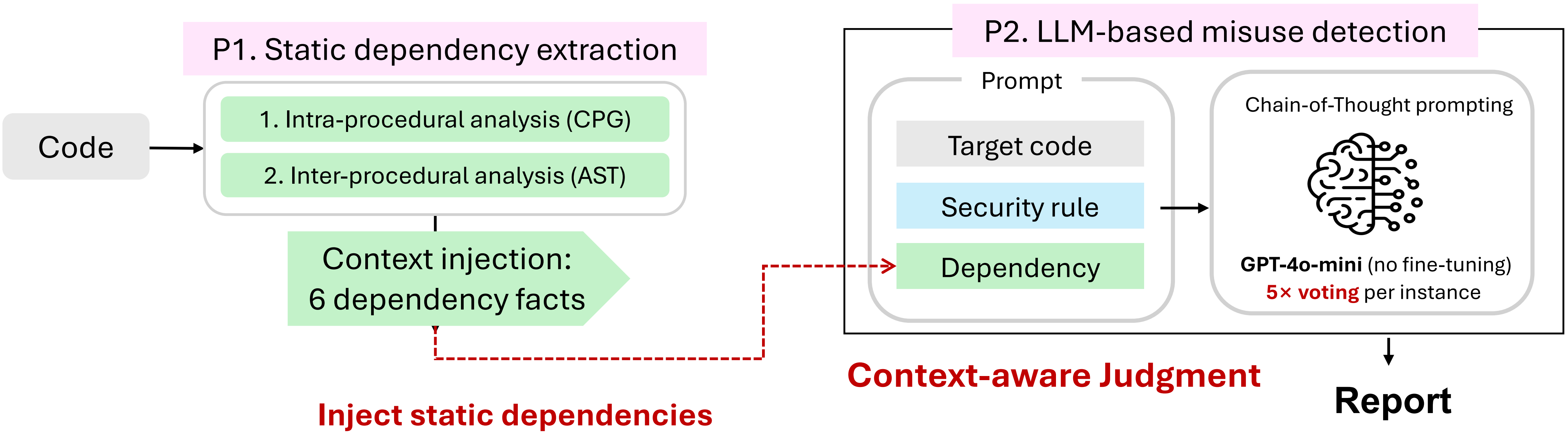
CRYPTBARA

Python CRYPTographic API misuse BARricAde

: Dependency-guided LLMs for precise Python crypto-API misuse detection

Design of CRYPTBARA

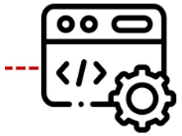
Dependency-guided LLM detection



P1. Static Dependency Analysis

Extract the facts LLM needs: who calls what, with which values, and where they come/go

INPUT



Target project

Identified cryptographic API calls

1. Intra-procedural analysis

- Receiver object
- Return value
- Call hierarchy

2. Inter-procedural analysis

- Parameter propagation
- Constant usage
- Call chain

Dependency information

P1-1. Intra-procedural analysis

Inside a function: receiver, return, call order

Item	Description	How it's traced
Receiver object	Object that calls the API	**CPG backward slice**
Return value	Variable holding API output	**CPG forward slice (1 hop)**
Call hierarchy	Who calls this API function	**Intra-procedural call scan**

P1-1. Intra-procedural analysis

Inside a function: receiver, return, call order

1. Receiver object : identify the object that calls the API

- How*:* CPG backward slice to the constructor/assignment

```
from Crypto.Cipher import AES

def encrypt_data(key, plain_text):
   cipher = AES.new(key, AES.MODE_ECB)
   encrypted_data = cipher.encrypt(plain_text)
   return encrypted_data

def handle_request():
   key = b'0123456789'
   plain_text = b'attack'
   enc_result = encrypt_data(key, plain_text)
   print(enc_result)
```

(1) Receiver object

Which object is initialized?

P1-1. Intra-procedural analysis

Inside a function: receiver, return, call order

2. Return value: capture the output variable and its next hop

- How: CPG forward slice (1 hop) to {return | store | arg}

```
from Crypto.Cipher import AES

def encrypt_data(key, plain_text):
  cipher = AES.new(key, AES.MODE_ECB)
  encrypted_data = cipher.encrypt(plain_text)
  return encrypted_data

def handle_request():
  key = b'0123456789'
  plain_text = b'attack'
  enc_result = encrypt_data(key, plain_text)
  print(enc_result)
```

(2) Return value

Which value is returned?

P1-1. Intra-procedural analysis

Inside a function: receiver, return, call order

3. Call hierarchy: list direct callers of this function

- How: intra-procedural call scan (caller → callee)

```
from Crypto.Cipher import AES

def encrypt_data(key, plain_text):
  cipher = AES.new(key, AES.MODE_ECB)
  encrypted_data = cipher.encrypt(plain_text)
  return encrypted_data

def handle_request():
  key = b'0123456789'
  plain_text = b'attack'
  enc_result = encrypt_data(key, plain_text)
  print(enc_result)
```

(3) Call hierarchy

Which function calls this?

P1-2. Inter-procedural analysis

Across functions: parameters, constants, call paths

Item	Description	How it's traced
Parameter propagation	Cross-function value flow	AST backtrace (follow caller→callee)
Constant usage	Hard-coded literals	AST literal scan → find literals → bind to arg → tag role & location
Call chain	End-to-end callers to API	Call-graph expansion (entry→API)

P1-2. Inter-procedural analysis

Across functions: parameters, constants, call paths

1. **Parameter propagation:** identify how security-sensitive values travel across functions

- **How:** AST back-trace at each call site, then follow caller→callee links to rebuild the cross-function flows

```
from Crypto.Cipher import AES
import hashlib

SALT = b'12345678'

def derive_key(password):
  global SALT
  key = hashlib.pbkdf2_hmac('sha256', password, SALT, 1000)
  return key

def encrypt_data(key, plain_text):
  cipher = AES.new(key, AES.MODE_ECB)
  return cipher.encyrpt(plain_text)

def handle_request():
  password = b'secret'
  key = derive_key(password)
  plain_text = b'Encrypt me'
  padded = plain_text.ljust(16, b'\x00')
  encrypted = encrypt_data(key, padded)
```

(1) Parameter propagation

Which value flows across functions?

P1-2. Inter-procedural analysis

Across functions: parameters, constants, call paths

2. **Constant usage:** detect hardcoded literals used as crypto API arguments
   - **How:** AST literal scan at call sites → find literals → bind each to the reached API arg → tag role & location

```
from Crypto.Cipher import AES
import hashlib

SALT = b'12345678'

def derive_key(password):
  global SALT
  key = hashlib.pbkdf2_hmac('sha256', password, SALT, 1000)
  return key

def encrypt_data(key, plain_text):
  cipher = AES.new(key, AES.MODE_ECB)
  return cipher.encyrpt(plain_text)

def handle_request():
  password = b'secret'
  key = derive_key(password)
  plain_text = b'Encrypt me'
  padded = plain_text.ljust(16, b'\x00')
  encrypted = encrypt_data(key, padded)
```

(2) Constant

Which constant is used?

P1-2. Inter-procedural analysis

Across functions: parameters, constants, call paths

3. **Call chain**: enumerate end-to-end callers that trigger the crypto API
    - **How:** build a call graph from caller→callee pairs and expand recursively from entry points to the API site

```
from Crypto.Cipher import AES
import hashlib

SALT = b'12345678'

def derive_key(password):
  global SALT
  key = hashlib.pbkdf2_hmac('sha256', password, SALT, 1000)
  return key

def encrypt_data(key, plain_text):
  cipher = AES.new(key, AES.MODE_ECB)
  return cipher.encyrpt(plain_text)

def handle_request():
  password = b'secret'
  key = derive_key(password)
  plain_text = b'Encrypt me'
  padded = plain_text.ljust(16, b'\x00')
  encrypted = encrypt_data(key, padded)
```

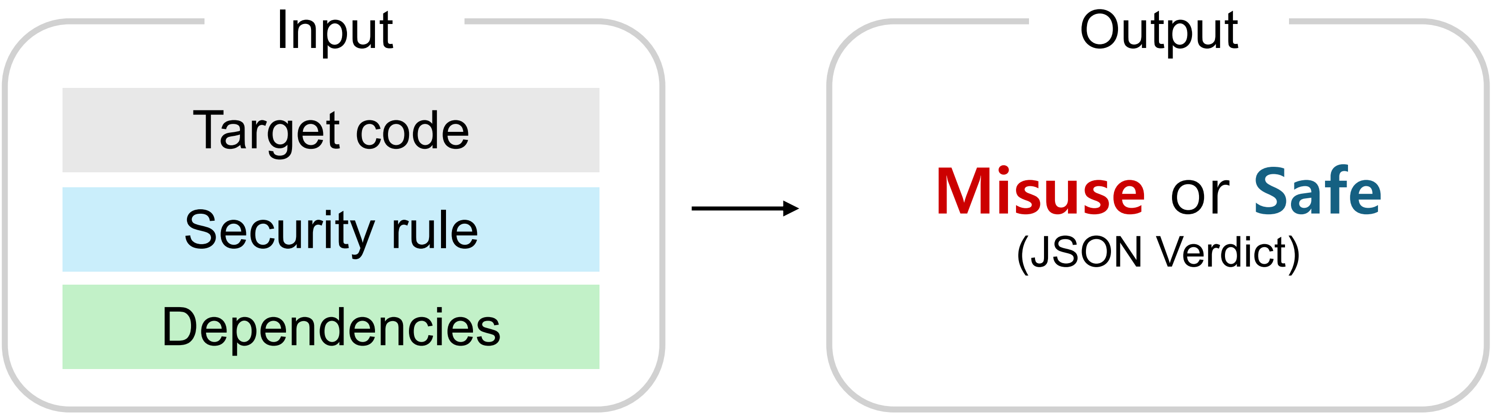
(3) Call chain

Which path leads to the API call?

P2. LLM-based Detection

Rule-scoped, dependency-guided LLM judging

- We use structured **rules** + **static dependencies** to bound the LLM's reasoning
- Inputs = *(target code, selected rule, dependencies)* → Output = JSON verdict
- **Per-rule** decision only (no out-of-scope critiques)



Rule

TABLE II: Summary of grouped cryptographic API misuse rules.

Group	ID	Category	Rule name	Checkpoints	Severity
Algorithm selection	R1	Symmetric encryption	Use secure and modern symmetric ciphers	Secure algorithm, sufficient key size	High
	R2	Asymmetric encryption	Use strong asymmetric key sizes	RSA $\geq$ 2048 bits, ECC $\geq$ 256 bits	High
	R3	Hash function	Avoid weak hash functions	Use SHA-256 or higher	High
	R4	Mode of operation	Avoid insecure block cipher modes	Use authenticated or randomized modes	High
Key and randomness management	R5	Key management	Avoid hardcoded or static keys	Keys should not be constant or predictable	High
	R6	PRNG quality	Use cryptographically secure PRNGs	Avoid `random` for secure keys	High
	R7	Seed management	Avoid predictable PRNG seeds	Use entropy-based seeding	Medium
Parameter and component management	R8	IV management	Avoid static IVs	IV should be randomized for each use	High
	R9	Salt management	Avoid static salts in PBE	Salt should be unpredictable	High
	R10	PBE iteration count	Use sufficient iteration count for PBE	#Iteration $\geq$ 100,000	High
Protocol and configuration security	R11	Secure configuration mode	Use authenticated cipher modes	Include MAC or AEAD mode	Medium

Evaluation

Accuracy: How precise is CRYPTBARA?

- Compare against **LICMA** and **Cryptolation**.

  Benchmarks : PyCryptoBench, Real-world set

  - PyCryptoBench (public benchmark; labeled crypto snippets)
  - Real-world set (curated real code; commit-level ground truth)

- CRYPTBARA outperformed prior tools
  - **Outperformed prior tools: 95.43% F1** on PyCryptoBench
  - **State-of-the-art on real code: 84.00% F1** on real-world set

TABLE III: Accuracy evaluation results on *PyCryptoBench*.

IDX	Group*	Tool	#TP	#FP	#FN	#TN	Precision	Recall	F1 score
	Total results	*LICMA*	10	**0**	86	**540**	**100.00%**	10.42%	18.87%
		Cryptolation	56	**0**	40	**540**	**100.00%**	58.33%	73.68%
		CRYPTBARA	**94**	7	**2**	533	93.07%	**97.92%**	**95.43%**

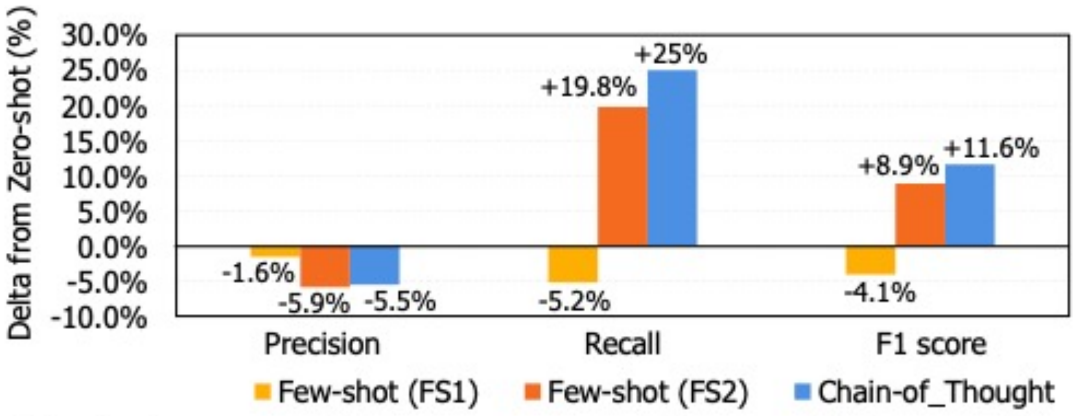
TABLE IV: Accuracy evaluation results on *real-world dataset*.

IDX	Group	Tool	#TP	#FP	#FN	#TN	Precision	Recall	F1 score
	Total results	*LICMA*	0	0	26	21	0.00%	0.00%	0.00%
		Cryptolation	18	7	8	14	72.00%	69.23%	70.59%
		CRYPTBARA	**21**	**3**	**5**	**18**	**87.50%**	**80.77%**	**84.00%**

Evaluation

Effectiveness: What makes CRYPTBARA accurate?

1. Prompt Design Comparison



2. LLM Backend Comparison

TABLE VI: Accuracy evaluation across different LLM backends.

Group*	Backend	#TP	#FP	#FN	#TN	Precision	Recall	F1 score
G1-1	GPT-3.5-turbo	**48**	**6**	**0**	**462**	**88.89%**	**100.00%**	**94.12%**
	LLaMA3	5	10	43	458	33.33%	10.42%	15.87%
	GPT-4o-mini	**48**	**6**	**0**	**462**	**88.89%**	**100.00%**	**94.12%**
G1-2	GPT-3.5-turbo	7	**0**	17	**12**	**100.00%**	29.17%	45.16%
	LLaMA3	17	**0**	7	**12**	**100.00%**	70.83%	82.93%
	GPT-4o-mini	**24**	1	**0**	11	96.00%	**100.00%**	**97.96%**
G1-3	GPT-3.5-turbo	0	**0**	24	**60**	0.00%	0.00%	0.00%
	LLaMA3	7	1	17	59	87.50%	29.17%	43.75%
	GPT-4o-mini	**22**	**0**	**2**	**60**	**100.00%**	**91.67%**	**95.65%**
Total	GPT-3.5-turbo	55	**6**	41	**534**	90.16%	57.29%	70.06%
	LLaMA3	29	11	67	529	72.50%	30.21%	42.65%
	GPT-4o-mini	**94**	7	**2**	533	**93.07%**	**97.92%**	**95.43%**

*We use the same group indices (IDX) as defined in Table III.

Rules + Dependency Facts + CoT prompting + GPT-4o-mini = 95.4% F1

Evaluation

Practicality: Can CRYPTBARA detect real-world threats?

- **Scope** = GitHub repos (★ ≥ 5,000)
- **Findings** = 172 potential misuses across 34 repos
- **Reporting** = 22 cases reported

→ **4 fixed**, **11 in discussion**, **7 low-risk (won't fix)** *(as of Aug 2025)*

Conclusion

- Python crypto API misuse is context-dependent
- **CRYPTBARA**
  - A **hybrid approach**: Static **Dependency** Analysis + **LLM reasoning** with rule-guided prompts
  - Turns raw code into **structured context** so the LLM can judge usage accurately
- **Effectiveness**
  - **Outperforms** state-of-the-art detectors in our evaluation
  - **172** previously unknown misuses discovered; **22** cases confirmed by developers
- CRYPTBARA helps ensure the secure and correct use of cryptographic libraries in Python

Thank you!

Contact

Seogyeong Cho jsg8777@korea.ac.kr

Software Security and Privacy Lab https://ssp.korea.ac.kr/